Noname manuscript No.
(will be inserted by the editor)

A Distributed Hybrid Index for Processing Continuous Range Queries over Moving Objects

Ziqiang Yu, Fatos Xhafa, Yuehui Chen, Kun Ma ✉

Received: date / Accepted: date

Abstract Central to many location-based services is the problem of processing concurrent continuous range queries over a large scale of moving objects. Most relevant works to this problem mainly investigate the centralized search algorithms based on a single server for handling range queries. However, due to the limited resources of a single server, these algorithms hardly can deal with an ocean of objects and extensive concurrent queries. Moreover, these approaches usually suppose either objects or queries are static but seldom consider the scenario that objects and queries are both moving simultaneously, restricting the practicability of these approaches.

To resolve the above issues, we propose a Distributed Hybrid Index (DHI) that consists of a global grid index and extensive local VR-tree indexes. DHI is apt to be deployed on a cluster of servers, and owns a good scalability to maintain numerous moving objects and concurrent range queries. Based on DHI, we further design a Distributed Incremental Search (DIS) approach, which organizes multiple servers with a publish/subscribe mechanism to calculate and monitor the results for continuous range queries in a distributed pattern. Finally, we conduct extensive experiments to fully evaluate the performance of our paper.

Keywords continuous range query, distributed hybrid index, incremental search.

1 Introduction

Nowadays, location-based services have become an important issue of service economy [8] [9]., and the processing of range queries over moving objects is a basic problem for

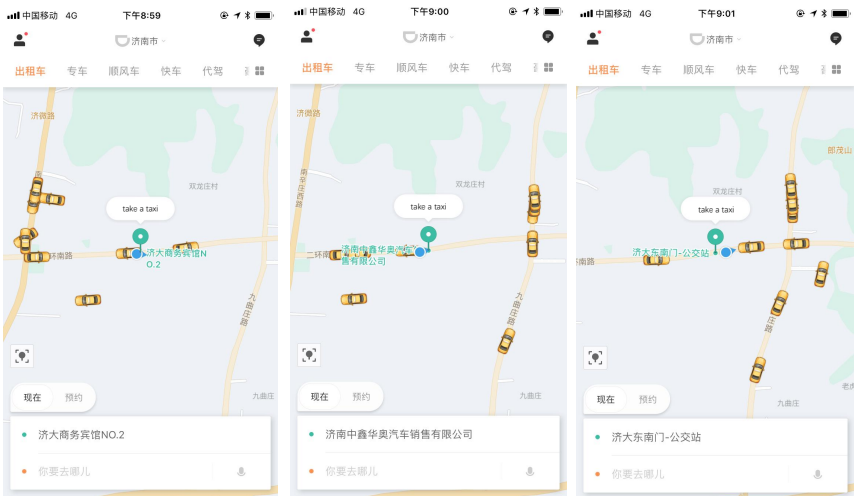
Ziqiang Yu
University of Jinan, Shandong Province, China, 250022
E-mail: ise_yuzq@ujn.edu.cn

Fatos Xhafa
Technical University of Catalonia, Barcelona, Spain

Yuehui Chen
University of Jinan, Shandong Province, China, 250022

Kun Ma (Corresponding author)
University of Jinan, Shandong Province, China, 250022
E-mail: ise_mak@ujn.edu.cn

many location-based services. In reality, various types of intellegent mobile terminal (e.g., sharing bycicles) in reality can be viewed as an moving object, which makes it urgent to study the Continuous Range Queries (CRQ) over moving objects. The processing of CRQ in this paper refers to calculate the result of a given query and then continuously monitor its result under the condition the queries and objects are simultaneously moving. This scenario enhances the complexity of this problem but it better accords with the reality. For example, a comprehensive service in a cab-hailing application (e.g., DiDi[1]) is that the data center needs to continuously seek for nearby taxies for a walking user who is using this application, where the user and taxies can be regarded as a CRQ and moving objects respectively. Due to the application needs to constantly update the taxies residing in the specified region centered on the user, so its essence is a CRQ over moving objects. Fig. 1 displays some screen captures of a cab-hailing application-DiDi.



(a) (b) (c)

Fig. 1 An example of a cab-hailing application-DiDi. *Fig.1(a), Fig.1(b), and Fig.1(c) represent the taxies in the rectangle centered on the user at different time points. With the position of user varying, the nearby taxies are also upated in real-time.*

Consider a set of moving objects in a two dimensional region of interest, the processing of a query q_i with the search scope sq_i is first to identify the objects covered by sq_i immediately, and then uninterruptedly monitor the result of q_i until it becomes invalid. In this process, once the movement of an object or q_i causes the change of the query result, we need to evaluate the new result of q_i in real-time. That is, every movement of an object or the query is deemed as a basic event that will trigger an evaluation on the result of this query. In this paper, we make no assumption on the motion of objects and adopt the snapshot semantics. In this semantics, the answer to q_i at time t_i is only valid for the past snapshot of the objects and q_i at time $t_i - \Delta t$, where Δt is the delay due to query processing. In order to guarantee the validity of answers, we endeavour to reduce Δt with operating all calculations in memory.

[1] http://www.didichuxing.com/en/

Be different with our definition of CRQ, most of the existing works [16][11] usually suppose that either the objects or the queries are static, which greatly reduces the complexity of this problem. Moreover, most existing proposals [2][5] pay more attention on centralized search algorithms based on a single server with assumption that all objects can be upload into the memory of one server. But in reality, the tremendous volume of moving objects and queries have been far beyond the capacity of a single server. For instance, the number of Wechat users reaches 600 million and every user can be viewed as a moving object or a query. For the sake of these reasons, it is tough to directly employ the existing methods to process CRQs over numerous moving objects.

In this paper, we strive to study a distributed scalable approach that can be suitably deployed on a cluster of servers to gain enough computing and storage capabilities. However, it is not a trivial problem to process CRQs in a distributed pattern because the following challenges must be addressed. Due to the objects and queries are distributed in different servers, so we first have to settle the problem of making multiple servers cooperatively compute the result for the same query in a parallel way. Additionally, the result of every CRQ need to be continuously monitored as the objects and queries are simultaneously moving, but we cannot recompute result for every CRQ once the movement occurs to avoid an unaffordable cost. Hence, we need to design a distributed incremental search approach to continuously compute the current result of a CRQ by reusing its previous result as far as possible. This work is certainly significant.

To meet above challenges, we construct a Distributed Hybrid Index (DHI) consisting of a global grid index [17] and extensive VR-tree indexes. In DHI, we employ the gird index to partition the whole region into numerous cells that are distributed in different servers, and further design a Variant-R-tree (VR-tree) to index the objects and CRQs in each cell. With the support of DHI, we further propose a Distributed Incremental Search (DIS) approach being apt to run on a cluster of servers. Specifically, we first compute the initial results for a CRQs in a distributed way, and then organize different servers with a publish/subscribe mechanism to cooperatively monitor and update the result of this *CRQ*. With the help of DHI and DIS, we are capable of processing CRQs efficiently when the objects and queries are both moving.

Our main contributions can be summarized as follows:

- We propose DHI, a distributed hybrid index that can be deployed on a cluster of servers to better support the distributed processing of CRQs than existing indexes.
- We adopt the publish-subscribe mechanism to organize multiple servers for processing CRQs in a parallel way.
- We develop DIS approach that can continuously evaluate the result of a CRQ by reusing the previous result of this CRQ as far as possible.
- We deploy our approach on S4 and implement two baseline approaches in expriments, and then evaluate the performance of our approach by comparing it with the baseline methods.

2 Related work

The processing of CRQ has broad application base, and it has been extensively studied by existing works. Some works [11][16] investigate this problem based on the road network. Stojanovic et.al. [11] proposes a framework for processing continuous range query when objects moving on road network, and then designs main-memory data structures to support reevaluation of continuous range queries. The work [16] processes range queries with the

Voronoi Range Search algorithm based on the Voronoi diagram. However, determining the Voronoi diagram for each object is very expensive, so this approach hardly can support range queries on frequent moving objects. Additionally, some works introduce the concept of safe region to reduce the reevaluation cost. Thereinto the paper [5] points out that the cost of monitoring and keeping the location of a moving query is very high, and it investigates an efficient technique by adopting the safe region. That is, as long as the query remains inside its specified safe region, expensive re-computation can be reduced. Cheema et.al [2] also adopt the methodology that utilizes the safe zone to monitor moving circular range queries and propose powerful pruning rules for improving the query efficiency. The above works mainly focus on the exact results of range queries, but the work [6] pays more attention to the approximate range search and proposes an approximate static range search algorithm. The work [12] coins a term "Region Queries" to indicate a broad category of spatial range queries, and aims to show a complete picture of region queries.

Recently, the increasing availability of indoor positioning, driven by technologies like Wi-Fi, RFID, and Bluetooth, enables a variety of indoor location-based services (LBSs). To support LBSs, some works [15][14][10] study a special range query, i.e., the indoor distance-aware queries on indoor moving objects. Xie et. al. [15] define and categorize the indoor distances in relation to indoor uncertain objects, and derive different distance bounds that can facilitate query evaluation. Shao et. al. [10] point out that the existing indexing and query processing techniques for the indoor space do not fully exploit the properties of the indoor space. Hence, they propose two novel indexes called Indoor Partitioning Tree (IP-Tree) and Vivid IP-Tree (VIP-Tree) by sufficiently considering the properties of indoor venues.

The above works present some excellent search algorithms for monitoring (continuous) range queries, but most of them are centralized and their scalabilities are restricted. Therefore, it is imperative to utilize a distributed computing model to deal with continuous range queries over numerous moving objects. Although some works[13][1][3][7] have explored this problem, but these approaches are hard to be applied in reality. This is because the distributed framework in these approaches consists of a central server and extensive moving objects, and these approaches mainly concern about how to transfer the work from the central server to the moving objects. So this type of approaches will consume amount of network band-with and the moving devices are required to own considerable computational capabilities, which restricts the applicability of these approaches. In contrast, our approach has no high demand on the computing power of mobile objects other than reporting their positions, which is an easy task for most moving objects. Therefore, our approach has a wider applicability.

3 Preliminaries

We illustrate some definitions and symbols utilized in later. The symbols and their meanings are summarized in Table 1.

Definition 1 ***(Object)*** A moving object o_i is represented by a triple, $\{t_c,(x_i,y_i),(x_i',y_i')\}$, where t_c is the current time and (x_i,y_i) and (x_i',y_i') separately represent the current and previous positions of o_i.

Definition 2 ***(Continuous Range Query, CRQ)*** A CRQ q_i is a triple $\{t_{q_i}^s,t_{q_i}^e,sq_i\}$, where $t_{q_i}^s$ and $t_{q_i}^e$ are the start and end time of q_i, and sq_i represents the search scope of q_i.

To facilitate present our work, we suppose sq_i is a rectangle hereinafter. In fact, our approach can support arbitrary shape of sq_i as long as it can be formulated by a polynomial.

Definition 3 ***(Processing of a CRQ)*** For a given CRQ q_i, the search process refers to compute the initial result of q_i at the time point $t^s_{q_i}$, and then continuously monitor and update the result of q_i until the time point $t^s_{q_i}$.

Definition 4 ***(Cell)*** In DHI, we utilize a grid index to partition the whole search region into numerous cells with the same size. A cell c_i is able to contain any number of objects.

Definition 5 ***(Obsolete-cell)*** For an object o_i, if a cell c_i covers its previous position (x'_i, y'_i), that is, $(x'_i, y'_i) \prec c_i$, then c_i is the *obsolete-cell* of o_i.

Definition 6 ***(Current-cell)*** For any object o_i, if the cell c_j covers its current position (x_i, y_i), that is, $(x_i, y_i) \prec c_j$, then c_j is the current-cell of o_i.

Definition 7 ***(Candidate-cell)*** For a CRQ q_i, if the cell c_k is fully or partially covered by sq_i, that is, $c_k \models sq_i$ or $c_k \vdash sq_i$, then c_k is a candidate-cell of q_i.

Since we aim to processing CRQs over moving objects in a distributed environment, the index structures and search algorithms in our proposal are all designed based on a Distributed Scalable Streaming Processing (DSSP) model. The principle of this model is presented as follows.

DSSP adopts the Actor model, and data flow within this model as events. An event takes the form of $\langle type, key, value \rangle$, and the basic logical computing unit is called a Processing Element (PE). A PE can selectively consume events output from other PEs, and emit new events that contain intermediate computing results to other PEs. PEs are designed to be very light-weight; thus one physical server can run thousands of PEs.

Due to the principle of DSSP is also adopted by some typical streaming platforms (such as S4, Storm), so our proposal also can be seamlessly transferred on these platforms.

Table 1 Summary of symbols

Symbols	Meaning
q_i	a continuous range query
sq_i	the search scope of q_i
$o \prec B$	an object o is covered by the region B
$o \not\prec B$	an object o is not covered by the region B
$C \bigotimes D$	the region C is has no intersection with the region D
$C \vdash D$	the region C is partially covered by the region D
$C \models D$	the region C is completely covered by the region D
$H - Z$	the objects belong to the set H but do not belong to the set Z

4 Distributed Hybrid Index

4.1 The structure of DHI

DHI consists of a grid index and extensive VR-tree indexes. The grid index serves as a global index structure that partitions the whole interest region into extensive four-square cells with the equal size, it is just maintained by a specified server. In order to void a bottleneck, we

make the grid index do not keep any object but only record the identifiers and boundaries of each cell. The cells are distributed in different severs. For a cell c_i, it owns three lists including OL_i, FL_i, and PL_i, where OL_i stores the moving objects in c_i, FL_i keeps the CRQs with search scopes fully covering c_i, and PL_i records the CRQs partially overlapping with c_i. Fig. 2 displays a DHI, where the cell c_i covers objects $(o_1, o_2, \cdots, o_5)$ and intersects with q_1, q_2, and q_3, so OL_i is $\{o_1, o_2, \cdots, o_5\}$, FL_i is $\{q_3\}$, and PL_i is $\{q_1, q_2\}$.

In reality, the cell c_i probably contains a large number of moving objects. Even if we determine that c_i is a candidate cell of a CRQ q_j, identifying the objects covered by sq_j from c_i is still a time-consuming task if c_i is partially covered by sq_j. To accelerate this process, we design VR-tree as a local index to maintain the moving objects and CRQs for every cell. The VR-tree is a variant of R-tree [4], but it is deeply optimized for processing CRQs. To alleviate the maintenance cost of VR-tree, we simplify its structure. Specifically, a VR-tree has only three levels. The first level is its root, the intermediate nodes in the second level are the children of the root, and the third level is the leaves. For a VR-tree T_i, its root refers to the whole region of the matching cell, every intermediate node corresponds to a smaller rectangle region, and the leaves are the smallest rectangles that store the moving objects. Fig.5 displays a VR-tree of the cell c_i in Fig. 4.

In comparison to R-tree, VR-tree has its own characteristics. First, the fixed depth not only can satisfy the requirement of improving the range query search efficiency but also reduces the maintenance expenses. Second, the leaves of VR-tree are the nodes keeping the moving objects rather than the real data in R-tree, restricting the quantity of leaves. Last but not the least, every leaf in VR-tree is allowed to store any number of objects and will no longer be removed or adjusted once it is created, which can further cut down its maintenance cost.

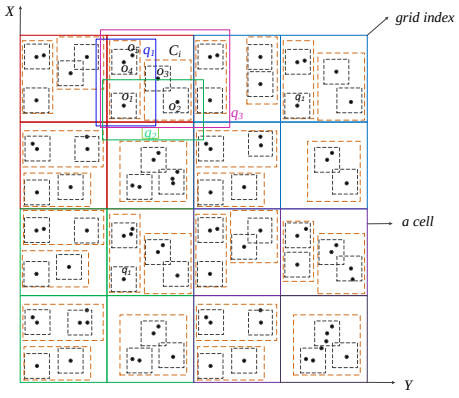


Fig. 2 The structure of DHI

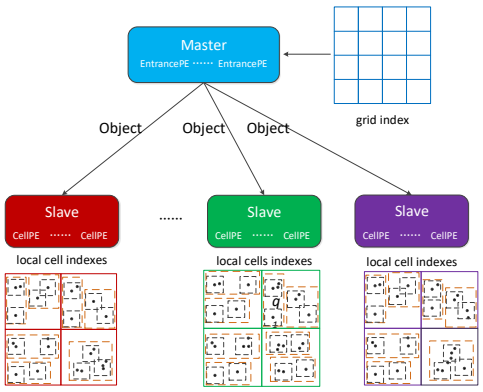


Fig. 3 Deploying DHI on the DSSP model

DHI is customized for the DSSP model that organizes a cluster of servers with the Master-Slaves pattern, that is, one server is the Master and other servers are Slaves. Fig.3 demonstrates the framework of deploying DHI (in Fig. 2) on DSSP. In this framework, the Mater operates multiple EntrancePEs, and every EntrancePE keeps a copy of grid index and acts as an entrance of objects and CRQs. Based on the grid index, EntrancePE can rapidly identify the obsolete-cell and current-cell for every object as well as the set of candidate cells for any *CRQ*. The cells in DHI are distributed in different Slaves, where each Slave

runs extensive CellPEs and every cell and its matching VR-tree are maintained by an unique CellPE.

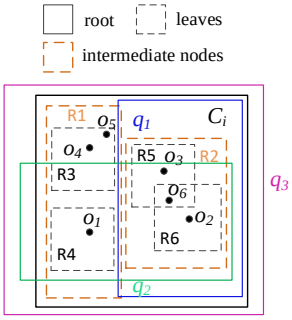


Fig. 4 A partitioned cell c_i

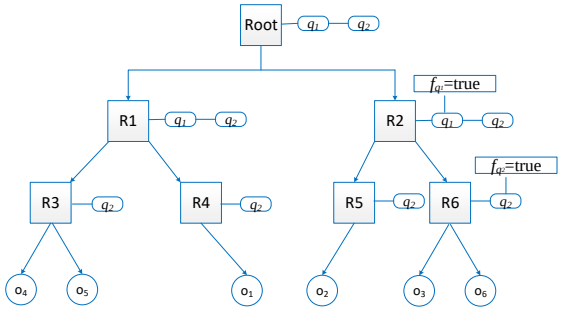


Fig. 5 A VR-tree corresponding to c_i

4.2 Construction of DHI

Building DHI refers to insert the objects and CRQs into different cells and their corresponding VR-trees.

4.2.1 The insertion of an object

For an object o_i $((x_i, y_i), (x_i', y_i'))$, we can easily identify the current-cell c_j of o_i, then o_i will be appended into OL_j and inserted into the VR-tree T_v corresponding to c_j. Algorithm 1 describes the process of inserting o_i into T_v. In this algorithm, we first scan T_v to identify a leaf that can cover the position (x_i, y_i), and then insert o_i into this leaf (lines 2-4). If such a leaf does not exist, we will create a new leaf n_l centered at o_i and insert o_i into n_l (lines 6-8). Subsequently, we need to adjust the VR-tree in terms of the creation of the new leaf (lines 9-17).

In VR-tree, we suppose every leaf has a unique number and their size are equal, which makes the construction of VR-tree more efficient. However, due to the sizes of leaves are identical, the new leaf will probably intersect with existing leaves to compose an superposed region. We use R_o and O_l to separately label the superposed region and the set of leaves forming this region. In this moment, if an object locates into the region R_o, we will appoint only the leaf with smallest number in O_l to keep it. This way, it is guaranteed that the leaves composing the superposed region will not contain the same object, which decreases the usage of the memory. In Fig.4, R5 and R6 form an overlapped region in the cell c_i. When the object o_6 entrances into this region, o_6 will be indexed by R5 if R5 has a smaller identifier than R6.

With an increasing number of objects being added into the VR-tree, more and more leaves will be created. Once the cell is completely covered by the leaves, no leaf will be created any more and the structure of the VR-tree will remain unchanged.

Algorithm 1 Inserting an object into VR-tree

Input: an object o_i, a VR-tree T_v
Output: the VR-tree T_v

```
1: Set a queue Q_p = ∅;
2: n_l=T_v.searchLeaf(o_i (x_i, y_i)) ;
3: if n_l exists then
4:     n_l.addObject(o_i (x_i, y_i));
5: else
6:     Create the new leaf n_l centered with o_i;
7:     n_l.addObject (o_i (x_i, y_i));
8:     n_w=n_l;
9:     while n_w ≠ r_t do
10:        Search a node n_p that can fully cover n_w from existing nodes of T_v;
11:        if n_p exists then
12:            Break;
13:        else
14:            Adjust or create an intermediate node n_p satisfying n_w ⊨ n_p;
15:            n_w=n_p;
16:        end if
17:    end while
18: end if
```

4.2.2 The insertion of a CRQ

To monitor CRQs in real-time, every cell is required to keep the CRQs partially overlapping with it, which can help us rapidly determine which cell will influence the result of a given CRQ. Therefore, we need to insert CRQs into DHI to make every cell index them. The insertion of a CRQ into a cell refers to add it into the corresponding VR-tree. For a CRQ q_j to be inserted, we appoint that if a node n_i of the VR-tree insects with the search scope sq_j, then n_i should record q_j because some objects in n_i may be covered by sq_j. In this case, we find that a node and its children will probably record multiple copies of the same query, which wastes the memory storage. For this, we introduce the *Query Indexing Rule* based on Theorem 1 to prevent the duplicate storage for the same CRQ. Particularly, we endow q_j a boolean attribute $f_{q_j}^{n_i}$, where n_i is a node in the VR-tree, and $f_{q_j}^{n_i} = false$ initially. If n_i is fully covered by sq_j, this rule specifies that q_j is only needed to be kept by n_i and any child of n_i does not record q_j. At this moment, we set $f_{q_j}^{n_i} = true$ after q_j being added to the node n_i to represent $n_i \models sq_j$. In fact, this strategy is not only able to reduce the copies of the same *CRQ* to be stored, but also can enhance the search efficiency. This is because all objects of the node n_i must belong to the result of q_j when $f_{q_j}^{n_i} = true$.

Theorem 1 *For a given CRQ q_i with sq_i to be inserted into a VR-tree T_v, if a node n_i in T_v satisfies $n_i \models sq_i$, then any child of n_i will be completely covered by sq_i,.*

Algorithm 2 shows the insertion of q_j into a VR-tree T_v. It first scans the VR-tree from the root with the width-first traversal algorithm (lines 4-15). When visiting a node n_i intersecting with sq_j, it asks n_i to keep q_j (lines 6-12). If $n_i \models sq_j$, then the children of n_i need not to be traversed (lines 7-9); otherwise, processing each child of n_i just like n_i. When all nodes of T_v are processed in this way, the insertion of q_j will be accomplished. Fig. 4 shows a cell c_i that has been partitioned to build a VR-tree, and Fig. 5 demonstrates the structure of the corresponding VR-tree. In this VR-tree, since R2$\models q_1$, so q_1 only needs to be kept by R2, while R5 and R6 has no need to record q_1.

Algorithm 2 Inserting a *CRQ* into VR-tree

Input: a *CRQ* q_j, the VR-tree T_v
Output: the VR-tree T_v

1: Set a queue $Q_n = \emptyset$;
2: r_t=the root of T_v;
3: Add r_t into Q_n;
4: **while** $Q_n \neq \emptyset$ **do**
5: n_i=Q_n.*removeFirstElement()*;
6: **if** $n_i \vdash sq_j$ **then**
7: n_i.addQuery (q_j);
8: **if** $n_i \models sq_j$ **then**
9: Set $f_{q_j}^{n_i} = true$;
10: continue;
11: **else**
12: Insert the children of n_i into Q_n;
13: **end if**
14: **end if**
15: **end while**

Algorithm 3 Deleting an object from VR-tree

Input: an object o_i, a VR-tree T_v
Output: the VR-tree T_v

1: $\mathscr{L}$ is a set of leaves;
2: $\mathscr{L}$=T_v.*searchLeaves* (o_i (x_i', y_i')) ;
3: **if** $|\mathscr{L}| > 1$ **then**
4: Sort leaves in $|\mathscr{L}|$ according to their numbers;
5: **for** leaf n_i: $\mathscr{L}$ **do**
6: **if** $o_i \prec n_i$ **then**
7: n_l=n_i;
8: **end if**
9: **end for**
10: **else**
11: n_l=$\mathscr{L}$.*getFirstElement()*;
12: **end if**
13: n_l.*remove* (o_i (x_i', y_i'));

4.3 The Maintenance of DHI

4.3.1 The maintenance of moving objects

When an object o_i ((x_i, y_i), (x_i',y_i')) arrives, we need to insert (x_i, y_i) into DHI and delete (x_i', y_i') from DHI in real-time. Since the boundaries of cells in the grid index are fixed, so the current-cell c_u and obsolete-cell c_o for o_i can be rapidly identified. Due to the insertion of (x_i, y_i) into the cell c_u can be achieved by Algorithm 1, we now only discuss how to remove (x_i', y_i') from the cell c_o, namely, deleting the previous position of o_i from the corresponding VR-tree of c_o with Algorithm 3. It first traverses T_v to identify the leaves probably covering the point (x_i', y_i'), and then it puts these leaves into a set $\mathscr{L}$ (line 1-2); If there exist more than one leaf in $\mathscr{L}$, which means (x_i', y_i') locates in an overlap region formed by multiple leaves. In this case, we will detect every leaf in an ascending order with respect to their numbers until discovering the leaf containing (x_i', y_i') (line 3-9); otherwise, we can directly remove (x_i', y_i') from the only one leaf in $\mathscr{L}$ (line 10). Due to the leaf of VR-tree can keep any number of objects and will not be adjusted once it is created, so the remove of an object will not change the structure of VR-tree.

4.3.2 The maintenance of CRQs

The real-time maintenance of *CRQ*s in DHI is a critical issue in our work because it will significantly affect the search efficiency and accuracy, and which will be more clear in next section. In DHI, the movement of a CRQ probably influence multiple cells, so we need to update the corresponding VR-trees to guarantee the accuracy of DHI when the CRQ occurs a movement. For a CRQ q_j, we suppose its search scope sq_j becomes sq'_j after one movement, and the set $\mathscr{C}$ includes the cells intersecting with sq_j or sq'_j. For any cell $c_i \in \mathscr{C}$, T_v is its matching VR-tree, the following cases need to be considered.

- Case1 $\{(c_i \bigotimes sq_j || c_i \vdash sq_j)$ and $c_i \models sq'_j\}$: if $c_i \bigotimes sq_j$ and $c_i \models sq'_j$, we only need to add q_j into the list FL_i; if $c_i \vdash sq_j$ and $c_i \models sq'_j$, we has to remove q_j from PL_i and insert it into FL_i. Addtionally, we need to remove q_j from T_v with Algorithm 4.
- Case2 $\{(c_i \vdash sq_j || c_i \models sq_j)$ and $c_i \bigotimes sq'_j\}$: if $c_i \vdash sq_j$ and $c_i \bigotimes sq'_j$, we delete q_j from the list PL_i and remove q_j from T_v; if $c_i \models sq_j$ and $c_i \bigotimes sq'_j$, we only need to remove q_j from the list FL_i.
- Case3 $\{(c_i \vdash sq_j || c_i \models sq_j)$ and $c_i \vdash sq'_j\}$: if $c_i \vdash sq_j$ and $c_i \vdash sq'_j$, q_j is still recorded in PL_i but we has to remove q_j from T_v based on sq_j and re-insert q_j into T_v on the basis of sq'_j; if $c_i \models sq_j$ and $c_i \vdash sq'_j$, q_j will be removed from FL_i and inserted into PL_i. Meanwhile, we need to insert q_j into T_v.

So far, we have presented a complete maintenance strategy of DHI and it has two benefits. First, we divide the whole maintenance of DHI into the small maintenance tasks based on every cell, and the maintenance of different cells can be processed simultaneously by multiple CellPEs in DSSP model, which can dramatically reduce the maintenance cost. Second, since the structure of VR-tree remains unchanged when no new leaf is created, so we do not need to consider how to adjust the VR-tree structure when updating the locations of objects and *CRQ*s in most times, which simplifies the maintenance of DHI.

Algorithm 4 Romoving a *CRQ* from VR-tree

Input: a *CRQ* q_j, a VR-tree T_v
Output: the VR-tree T_v

```
1: 𝒩 is a set of nodes and 𝒩 = ∅;
2: Inserting the root of T_v into 𝒩;
3: while 𝒩 ≠ ∅ do
4:     Node n_t=𝒩.getFirstElement();
5:     if n_t ⊢ sq_j or n_t ⊨ sq_j then
6:         Removing q_j from the query list of n_t;
7:         if n_t ⊢ sq_j then
8:             Put the children of n_t into 𝒩;
9:         end if
10:    end if
11: end while
12: return T_v;
```

5 Distributed Incremental Search of CRQs

In this section, we propose the DIS approach to compute and monitor the results of extensive CRQs in a distributed environment. For a given CRQ, DIS consists of two phases. In the first

phase, it introduces a Parallel Search Algorithm (PSA) based on the DSSP model to compute the initial result for every CRQ, and then designs a Distributed Incremental Continuous Monitoring (DICM) strategy to continuously update their results in the second phase.

5.1 Search the initial result for a CRQ

When a new CRQ q_j arrives, PSA first determines $\mathscr{F}_j$, the set of candidate cells of q_j based on the grid index. In PSA, we will search the objects covered by sq_j from each candidate cell simultaneously. Here, the major task is how to rapidly search the result from the cells partially overlapping with q_j. A naive method is detecting all objects in each cell, but it will cause a notable expense. In PSA, we employ the VR-tree to solve this problem and Algorithm 5 describes the search procedure. For a VR-tree T_v, we first traverses T_v to find Γ, the set of the leaves intersecting with sq_j, and then search the objects covered by sq_j from every leaf in Γ. This way, only the leaves intersecting with sq_j needs to be detected. Additionally, for any node n_i in T_v, if $f_{q_j}^{n_i} = true$, we can infer that all objects covered by n_i belong to the result of q_i, then the branch with n_i being a parent has no need to be processed. After all cells in Γ have been processed, PSA will combine the partial result generated by each candidate cell into the final result.

Theorem 2 *For a CRQ q_i and a cell c_j corresponding to a VR-tree T_v, if $c_j \vdash sq_i$, the time of searching objects covered by sq_i from c_j is about $\lceil (sq_i \cap s_c)/s_l \rceil \cdot N_l \cdot t_d$ on condition that the objects in c_j conform to a uniform distribution, where s_l is the size of a leaf in T_v, s_c is the size of the cell c_j, N_l is the average number of objects covered by a leaf, and t_d is the constant time of judging whether an object is covered by q_i.*

Proof Since the objects in c_j conform to a uniform distribution, then the number of leaves intersecting with sq_i in T_v is almost $\lceil (sq_i \cap s_c)/s_l \rceil$, where s_l is the size of a leaf. Meanwhile, we suppose the average number of objects contained by a leaf is N_l, then the number of object to be detected is about $\lceil (sq_i \cap s_c)/s_l \rceil \cdot N_c$. Therefore, the time of searching the result from the cell c_j with Algorithm 5 approximately equals to $\lceil (sq_i \cap s_c)/s_l \rceil \cdot N_l \cdot t_d$.

Algorithm 5 Searching the result of q_i from a cell c_j

Input: a CRQ q_j, a VR-tree T_v matching c_j
Output: the partial result of q_i

```
1: Set ℛ = ∅, 𝒩 = ∅, Γ = ∅;
2: Set r_v as the root of T_v;
3: Put the children of r_v into 𝒩;
4: while 𝒩 ≠ ∅ do
5:     Node n_t=𝒩.getFirstElement();
6:     if n_t.queryList contains q_i then
7:         if f_{q_i}^{n_t}=true then
8:             Put children of n_t into Γ;
9:         else
10:            Put children of n_t into 𝒩;
11:        end if
12:    end if
13: end while
14: for leaf l_i:Γ do
15:    Search objects covered by sq_i from l_i, and put these objects into R;
16:    ℛ=ℛ∪R;
17: end for
18: return ℛ;
```

The deployment of PSA on the DSSP model is shown in Fig. 6. In particular, when an EntrancePE receives a CRQ q_i, it first determines the set of candidate cells of q_i. Next, the EntrancePE will send a *queryEvent* carrying q_i to the candidate CellPEs, where each candidate CellPE corresponds to a candidate cell. Once a candidate CellPE receives the *queryEvent*, it will search the objects covered by sq_i from the corresponding cell as the partial result of q_i with Algorithm 5. The partial result will be encapsulated as a *resultParEvent* and it will be sent to a QueryPE. This QueryPE will maintain the results of q_i all the time until q_i becomes invalid. In our model, a query will be processed by an unique QueryPE but this QueryPE can deal with more than one query at the same time. In this scenario, if we set the identifier of q_i as the key of *resultParEvent*s of q_i, these *resultParEvent*s will be routed to the same QueryPE. After receiving all partial results generated by every candidate CellPE, the QueryPE can readily get the final result of q_i.

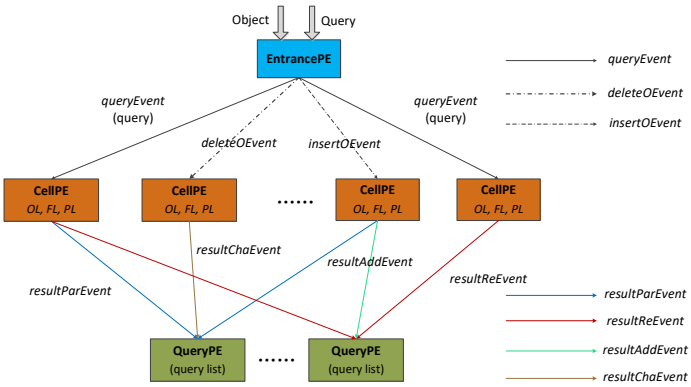


Fig. 6 Deploying DIS on DSSP model

5.2 Incrementally monitoring CRQs

After obtaining the initial results of CRQs, we still need to continuously update their results when the objects and queries are moving. Here, we face two difficult problems. The first one is how to rapidly detect which CRQs will be influenced by the movement of an object, and the second one is how to incrementally update the result rather than recompute the whole result for a CRQ when it moves. To address these issues, we design a DICM method that employs a publish/subscribe mechanism to make multiple servers cooperatively monitor CRQs in a parallel way. Similarly, DICM also adopts DSSP as the basic computing model.

In DICM method, CellPEs and QueryPEs are regarded as query result publishers and subscribers respectively, and their working schema is also displayed in Fig. 6. For the query q_i, we first talk about how to monitor its result as the objects move. After the initial result obtained, q_i will be registered in every candidate cell, that is, q_i will be added into the query list FL or PL of every candidate cell. Meanwhile, each candidate CellPE will simultaneously monitor the partial result of q_i. Once a candidate-CellPE discovers that a partial result of q_i changes because of the movement of an object, it will instantly package the changed result into a *resultChaEvent* and send this event to the QueryPE maintaining q_i, and the QueryPE will update the result of q_i when receiving the *resultChaEvent* as soon as possible.

Due to the result of q_i just concerns about the objects covered by sq_i rather than their exact positions, so we can utilize Theorem 3 to filter the CRQs that need not to be monitored. Particularly, a cell c_i only needs to monitor the CRQs in PL_i because the results of the CRQs in FL_i will not be affected by the movement of objects.

Theorem 3 *In a cell c_i, $\forall o_i \in OL_i$, if (x_i', y_i') and (x_i, y_i) are both covered by c_i, then $\forall q_j \in FL_i$, the result of q_j will not be influenced by the movement of o_i.*

Next we investigate how to update the result of a CRQ when it moves constantly. Here, we adopt the semantics that the result of a CRQ at any time t_i is calculated based on the snapshot of objects at the same time. Due to the maintenances of objects and CRQs are proceeded simultaneously, so the snapshot of objects can invariably include the latest locations of moving objects, which implies the result of a CRQ can be up to date at any time only if the computing time is small enough.

For the query q_i, if its search scope sq_i becomes sq_i' after a movement, it will be resubmitted to our model. When receiving the resubmitted q_i, EntrancePE will recompute its candidate cells based on sq_i', and then send q_i to the candidate-CellPEs. After receiving q_i, every candidate CellPE will updates two registered query lists of the matching cell right away, and then only calculates the partial changed result of q_i in a parallel way. Next, these candidate CellPEs send the changed results to the QueryPE taking charge of q_i, and this QueryPE will update the result of q_i by processing all changed results.

In above process, the critical task is maintaining the registered query lists and compute the changed results of q_i by the corresponding CellPEs. We suppose c_k is a candidate cell of q_i, PE_k^c is the candidate CellPE matching c_k, and PE_i^q is the QueryPE maintaining q_i , all cases need to be handled for processing q_i are summarized as follows.

(1) If q_i is a new registered query for c_k, there are two cases to be processed.

- If $c_k \models sq_i'$, PE_k^c will insert q_i into FL_k and package all objects of c_k into a *resultAddEvent* that will be sent to PE_i^q, while PE_i^q will add these objects into the result of q_i after receiving the *resultAddEvent*.
- If $c_k \vdash sq_i'$, PE_k^c will insert q_i into PL_k and find the objects covered by sq_i' with Algorithm 5. Likewise, these objects will be sent to PE_i^q as the incremental result of q_i.

(2) If q_i is an existed registered query of c_k, PE_k^c has to deal with the following situations.

- If $(sq_i \models c_k)$ && $(sq_i' \models c_k)$, which means q_i has been in FL_k and it has no need to be inserted again. Apparently, PE_k^c has no need to update the result of q_i in this case.
- If $(sq_i \vdash c_k)$ && $(sq_i' \models c_k)$, PE_k^c needs to remove q_i from PL_k and insert it into FL_k. Moreover, PE_k^c needs to identify the objects covered by the region $(sq_i' - sq_i)$ and inform PE_i^q to expand the result of q_i to include these objects by sending a *resultAddEvent*.
- If $(sq_i \models c_k)$ && $(sq_i' \vdash c_k)$, PE_k^c has to remove q_i from FL_k and insert it into PL_k. Additionally, PE_k^c needs to find the objects residing in the scope $(sq_i - sq_i')$ and send these objects as a *resultReEvent* to PE_i^q, while PE_i^q will remove the objects carried by the *resultReEvent* from the result of q_i.
- If $(sq_i \vdash c_k)$ && $(sq_i' \vdash c_k)$, q_i is still in PL_k but PE_k^c needs to update the search scope of q_i. Moreover, PE_k^c has to identify P_i and P_i', two sets of objects separately covered by two regions sq and sq', and notify PE_i^q to remove the objects belonging to $(P_i - P_i')$ from the result of q_i and add the objects in $(P_i' - P_i)$ into the result.

So far, we have presented the complete procedure of DICM. In this method, we organize extensive CellPEs and QueyPEs based on a publish/subscribe mechanism to continuously

monitor every CRQ in a parallel way, which makes DICM be able to take full advantage of a cluster of servers to handle CRQs and its capability can be dramatically improved employing more servers. Moreover, DICM adopts an incremental search strategy in which every CellPE is precisely aware of which CRQs need to be re-evaluated and only needs to calculate the changed part of the result for every CRQ. Finally, every CellPE adopting Algorithm 5 based on the VR-tree to filter the objects that need not to be detected.

6 Experiments

6.1 Experimental setup

We conduct experiments to evaluate the performance of DHI and DIS. Here, we introduce two other distributed algorithms as baseline methods to fully evaluate the performance of DSI. The first one is a Naive Search (NS) algorithm which does not use any index. NS will radomly send the objects to different servers, so it needs to scan all objects maintained by different servers for processing every CRQ. Another baseline algorithm is a Simplified Version of DIS approach (SV-DIS). Specifically, it only utilizes a distributed grid index without further building VR-tree for each cell when handling CRQs.

This experiment employs three simulate datasets based on a German road network. In the first dataset (UD), the objects follow a uniform distribution. In the second dataset (GD), 70% of the objects follow the Gaussian distribution, and the other objects are uniformly distributed. The objects in the third dataset conform to the Zipf distribution and is generated in the following way. We first rank the roads in descending order based on their length. For any road r_i, we use h_i to represent its rank and assign to it a probability $P(r_i) = C/h_i^{\alpha}$, where C and α are constants set as 0.001 and 0.9 respectively. Next, we let the percentage of objects appearing on road r_i be equal to $P(r_i)$, and the objects are randomly distributed on rising obviously.

The reason why we adopt three different types of datasets is to more sufficiently evaluate the performance of our approach and better certify its universality. As to many roads in reality, the distribution of vehicles is uniform, so the uniform distribution of moving objects in our experiments can simulate this scenario. The Gaussian distribution is widely used to approximate the data distribution in various fields, hence if our approach has a good performance on the Gaussian distribution, which certifies our method has wide applicability. In the city road network, we find that most vehicles are in a few major roads especially in the morning and evening rush hours. To simulate this phenomena, we construct the Zipf distribution of moving objects as a dataset in this part.

In these datasets, the whole interest region is normalized to a unit square. By default, the side length of a cell in DHI is set 0.01 and the side length of a leaf in VR-tree is set 0.002. Moreover, all objects move along the road network, with the velocity uniformly distributed in [0, 0.005]. We use V_o and V_q to represent the velocities of objects and queries respectively, and adopt o_v and q_v to label the arrival rates of objects and queries separately. The experiments are conducted on a cluster of 8 Dell R210 servers with Gigabit Ethernet interconnect.

6.2 Performance of DHI

We mainly evaluate the performance of DHI in this section. In the following experiments, the size of an object is about 60 bytes, then we has to handle approximately 30GB data when the number of objects takes the maximum value.

Time of build DHI. Fig. 7 gives the time of building DHI with the number of objects varying. Here, we feed the objects into the system one time and then test the time of building DHI for these objects. As the number of objects increases, every server generally needs to process more objects to build DHI, causing the time almost growing linearly with the rising number of objects. Additionally, the time is slightly higher for GD and Zipf, indicating that a more clustered distribution may lead to a higher cost of building DHI. The reason is that certain servers in the cluster probably need to handle more objects than other servers when objects gather more densely, and these servers with heavier workload impair the whole performance of DHI.

Memory consumption of DHI. Fig. 8 evaluates the approximate memory consumption of DHI with the number of objects varying. As expected, the growth of consumed memory is proportion to the volume of objects and the total used memory almost reaches 35GB when indexing 500M objects. Moreover, the results manifest the distribution of objects has little impact on the memory consumption.

Throughput of DHI. Throughput of DHI is an essence metric and we test it in Fig. 9. In this evaluation, we set a queue Q_o to cache the arriving objects and then make DHI constantly consume objects from Q_o. We set the capacity of Q_o as 100K objects and make the objects enter the system with varying arrival rates. Next, we monitor the occupied capacity of Q_o and display the results in 9. Observed from the results, Q_o always caches a small number of objects when the arrival rate is not greater than 3 million *Objects Per Second* (*OPS*), but the occupied capacity of Q_o increases rapidly after the arrival rate reaching 4 million *OPS*, which implies the throughput of DHI is about 4 million *OPS*.

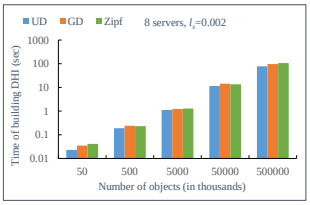


Fig. 7 Time of building DHI

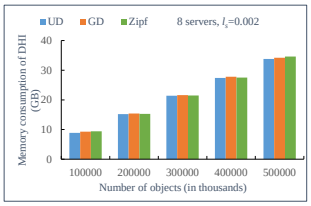


Fig. 8 Memory Consumption of DHI

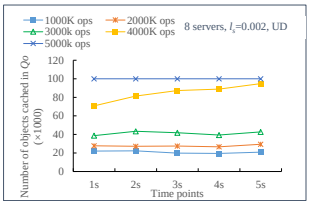


Fig. 9 Throughput of DHI

Effect of the velocity of objects. In Fig. 10, we first build DHI for 50M objects and then randomly select 5M objects from them. Next, we make these chose objects move ten times with the same velocity to test the average cost of maintaining DHI for processing these objects one time. Here, we conduct multiple tests corresponding to different velocities of objects. The results show that the maintenance cost obviously grows with the increasing velocity of objects when V_o is not greater than 0.003. After that, the maintenance cost almost remain unchanged. This is because when the moving velocity is lower , the obsolete and new positions of an object are more likely to locate in the same leaf of a VR-tree, then the time of identifying obsolete and new positions of this object in the VR-tree can be reduced by half.

Effect of *leaf-size* of VR-tree. *Leaf-size* determines the structure of the VR-tree, so it probably influences the performance of DHI. In our experiments, the *leaf-size* of a VR-tree is represented by the side length of a leaf. Fig. 11 demonstrates that the building time decreases obviously as the the side length grows regardless of objects conforming to which distribution. The reason is that a VR-tree will include less nodes if the side lengths of leaves become greater, and the time of building DHI can be cut down when the structures of extensive VR-trees become more simple.

Scalability of DHI. In Fig. 12, we evaluate the scalability of DHI by measuring the time of building DHI for 100M objects with varying number of servers. The left vertical axis represents the building time and it decreases sharply with the increasing number of servers, testifying DHI can scale well with adding servers into the computing cluster. Moreover, we make objects enter the system with the rate of 3000 *OPS*, and then evaluate the maximum occupied capacity of Q_o with a different number of servers. Based on the results shown as the right vertical axis, we clearly see that the maximum volume of cached objects in Q_o drops dramatically with more servers adopted, which indicates that the throughput of DHI can be easily extended by employing more servers.

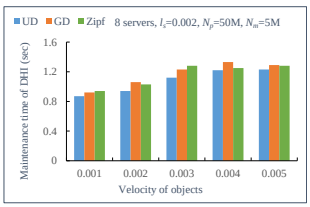


Fig. 10 DHI performance w.r.t. velocity of objects

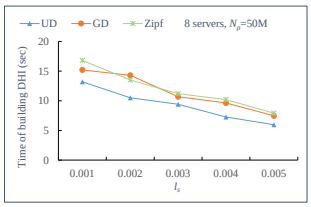


Fig. 11 DHI performance w.r.t. l_s

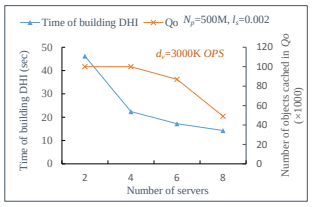


Fig. 12 Scalability of DHI

6.3 Comparison of three approaches

Comparison of response time. In this group of experiments, we generate a set of CRQs, where every CRQ has a circle search scope with the radius (r_q) being 0.015. In Fig. 13, we test the response time of three approaches when they compute the initial results for a varying number of CRQs. From the results, we clearly see that DIS always performs better than NS and S-DIS. This is because NS will make all servers process every query with a brute-force search pattern, which makes its response time grow linearly with number of CRQs. As to S-DIS, it usually needs to detect more objects than DIS for processing the same set of CRQs because it lacks the support of VR-tree.

Comparison of scalability. In Fig. 14, we employ a different number of servers to operate these three approaches and measure their response times for computing the initial results of 10000 *CRQ*s. The results clearly demonstrate that the response times of three approaches all decline significantly with more servers being employed, but the processing times of DIS and S-DIS drop more sharply with an increasing number of servers, indicating DIS and S-DIS own the better scalability than NS.

Comparison of incremental search time. In Fig. 15, we feed 2000 CRQs into the system and make them move 0.01 one time. In experiments, we record the times of processing these queries by three approaches at five consecutive time points. We observe that DIS performs better than other two approaches especially at the last four time points. For NS and S-DIS, they always process every query as a new one, which leads to their response times

almost remain unchanged at every time point. But for DIS, it only calculates the incremental results of the queries after obtaining the initial results, which can significantly reduce the computing time.

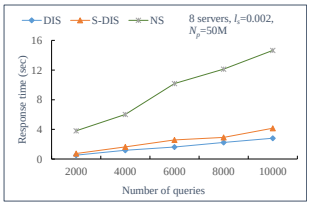


Fig. 13 Comparison of the response time

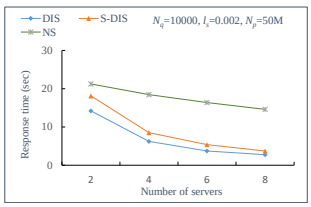


Fig. 14 Comparison of the scalability

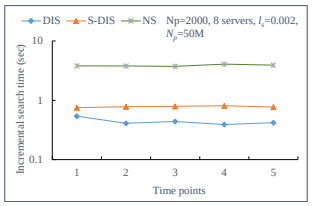


Fig. 15 Comparison of the incremental search time

Comparison of throughput. From Fig. 16 to Fig. 18, we evaluate the throughputs of three approaches by computing initial results of the given CRQs. In these experiments, we make CRQs enter the system with different arrival rates and then monitor the number of CRQs cached in Q_r at a sequence of time points, where Q_r is a queue to cache the arriving CRQs and its capacity is set to 10000 queries. From Fig. 16, we find that the occupied capacity of Q_r is always small when the arrival rate (q_v) is not greater than 4000 *Query Per Second* (*QPS*), but it starts to increase obviously as q_v reaches 5000 *QPS*. Hence, we conclude that the throughput of DIS is about 4000 *QPS*. From other two figures, we deduce that the throughputs of S-DIS and NS are approximately 3000 *QPS* and nearly 1000 *QPS* respectively, which are both smaller than that of DIS.

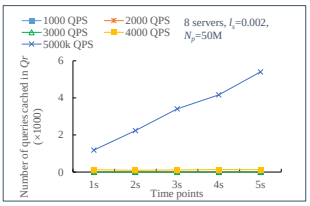


Fig. 16 Throughput of DIS

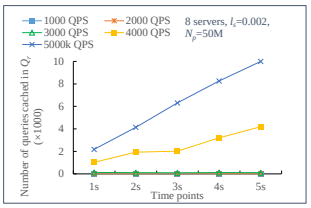


Fig. 17 Throughput of S-DIS

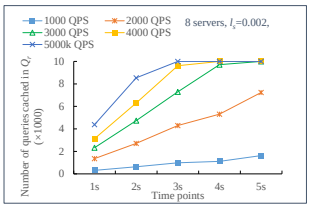


Fig. 18 Throughput of NS

Effect of number of objects. In Fig. 19, we evaluate the influence of the volume of objects on performance of three approaches. For every approach, we will put 10000 CRQs into the system and record the time between the first query entering the system and the results of all queries being obtained. With the number of objects increasing, the response time of NS grows in a linear pattern but the performances of S-DIS and DIS are almost not affected. This is because NS needs to scan all objects for processing each query, while S-DIS and DIS can effectively prune the search space, which makes them only need to process a restricted number of objects even if the volume of objects grows dramatically.

6.4 Performance of DIS

Effect of *leaf-size* of a VR-tree. In Fig. 20, we test the time of computing initial results of 10000 CRQs when the side length of the leaf in VR-tree adopts different values. The results

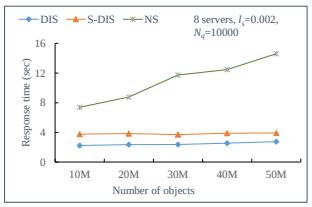


Fig. 19 Effect of the number of objects

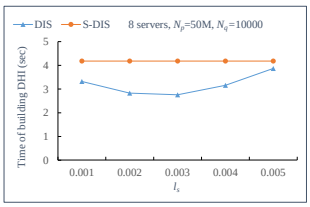


Fig. 20 Effect of *leaf-size* on DIS

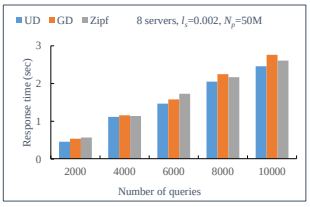


Fig. 21 Effect of number of queries on DIS

display that the search time first declines but then grows up with the side length increasing, where $l_s = 0.003$ is a turning point. When the side length is too small, the cost of searching leaves intersecting with a query will increase, and too small leaves hardly can support the pruning strategy based on VR-tree. Oppositely, if the side length becomes too large, more objects in a leaf then need to be detected, making the effect of filtering objects deteriorate seriously. Therefore, an appropriate side length can effectively refine the search space for processing CRQs.

Effect of the data distribution. In Fig. 21, we evaluate the influence of distributions of objects on the performance of DIS. To achieve this purpose, we make DIS search the initial results of 2000 CRQs based on three different distributions. As the number of queries increases, the response time of DIS grows gradually but the different distributions have little influence on the performance of DIS. This is because DIS can effectively prune the objects that need not to be detected for every CRQ even if the objects are more clustered.

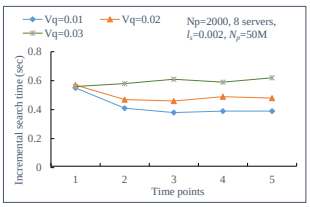


Fig. 22 Effect of velocity of queries on DIS

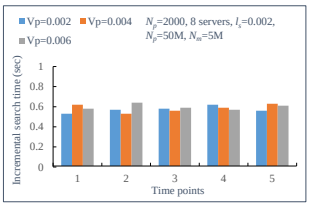


Fig. 23 Effect of velocity of queries on DIS

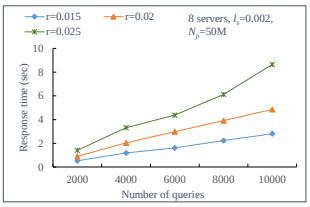


Fig. 24 Effect of search scope on DIS

Effect of the velocity of queries. In Fig. 22, we test the influence of the velocity of CRQs on the performance of DIS. We first put 2000 CRQs into the system and make them continuously move together with the same velocity. Next, we adopt DIS to process these CRQs once their movements occur and measure the response time of DIS at five consecutive time points. Finally, we conduct three group of tests to evaluate the impact of different velocities of CRQs on DIS. Comparing the costs of DIS at the same time point, we find that the higher velocity increases the processing time. Indeed, the incremental strategy introduced by DIS can prune the search space commonly covered by the previous and current query scopes of a CRQ. When the velocity is low, a larger space usually can be pruned to save the search time. As the velocity becomes higher, the overlapped space between two adjacent search scopes of a CRQ decreases and the pruning effect of DIS declines.

Effect of the velocity of objects. We also test the influence of the velocity of objects on the performance of DIS in Fig. 23. Due to the result of a query is calculated based on a

current snapshot of moving objects, so the velocity of objects has little affect on the response time of DIS and the experimental results also certify this point.

Effect of the search scope. Finally, we evaluate the relationship between the performance of DIS and the search scope of CRQs. In Fig. 24 , we process a varying number of CRQs with different sizes of search scopes. The results demonstrates that the search scope imposes a significant impact on the performance of DIS. Particularly, larger search scope of a CRQ will cause more CellPEs in different servers to process this query. In this process, the master probably needs to communicate with more servers and the growing communication cost magnifies the response time of DIS.

7 Conclusion

With the dramatic increase of mobile devices and the advances in wireless network, the efficient processing of CRQs has been of increasing interest. In this context, this paper proposes a distributed incremental search approach that sufficiently considers the situation where queries and objects are both moving, and only needs to reevaluate the incremental result of every CRQ as the objects and queries are moving. Finally extensive experiments are conducted to verify the performance of our approach.

References

1. Cai, Y., Hua, K.A., Cao, G.: Processing range-monitoring queries on heterogeneous mobile objects. In: Mobile Data Management, 2004. Proceedings. 2004 IEEE International Conference on, pp. 27–38. IEEE (2004)
2. Cheema, M.A., Brankovic, L., Lin, X., Zhang, W., Wang, W.: Multi-guarded safe zone: An effective technique to monitor moving circular range queries. In: Data Engineering (ICDE), 2010 IEEE 26th International Conference on, pp. 189–200. IEEE (2010)
3. Gedik, B., Liu, L.: Mobieyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In: International Conference on Extending Database Technology, pp. 67–87. Springer (2004)
4. Guttman, A.: R-trees: a dynamic index structure for spatial searching, vol. 14. ACM (1984)
5. Haidar, A.K., Taniar, D., Betts, J., Alamri, S.: On finding safe regions for moving range queries. Mathematical and Computer Modelling **58**(5), 1449–1458 (2013)
6. Haidar, A.K., Taniar, D., Safar, M.: Approximate algorithms for static and continuous range queries in mobile navigation. Computing **95**(10-11), 949–976 (2013)
7. Hu, H., Xu, J., Lee, D.L.: A generic framework for monitoring continuous spatial queries over moving objects. In: Proceedings of the 2005 ACM SIGMOD international conference on Management of data, pp. 479–490. ACM (2005)
8. Kaczor, S., Kryvinska, N.: It is all about services-fundamentals, drivers, and business models. Journal of Service Science Research **5**(2), 125–154 (2013)
9. Molnr, E., Molnr, R., Kryvinska, N., Gregu, M.: Web intelligence in practice. Journal of Service Science Research **6**(1), 149–172 (2014)
10. Shao, Z., Cheema, M.A., Taniar, D., Lu, H.: Vip-tree: an effective index for indoor spatial queries. Proceedings of the VLDB Endowment **10**(4), 325–336 (2016)
11. Stojanovic, D., Papadopoulos, A.N., Predic, B., Djordjevic-Kajan, S., Nanopoulos, A.: Continuous range monitoring of mobile objects in road networks. Data & Knowledge Engineering **64**(1), 77–100 (2008)
12. Taniar, D., Rahayu, W.: A taxonomy for region queries in spatial databases. Journal of Computer and System Sciences **81**(8), 1508–1531 (2015)
13. Wang, H., Zimmermann, R., Ku, W.S.: Distributed continuous range query processing on moving objects. In: International Conference on Database and Expert Systems Applications, pp. 655–665. Springer (2006)
14. Wang, Z.J., Yao, B., Cheng, R., Gao, X., Zou, L., Guan, H., Guo, M.: Sme: explicit & implicit constrained-space probabilistic threshold range queries for moving objects. GeoInformatica **20**(1), 19–58 (2016)

15. Xie, X., Lu, H., Pedersen, T.B.: Efficient distance-aware query evaluation on indoor moving objects. In: Data Engineering (ICDE), 2013 IEEE 29th International Conference on, pp. 434–445. IEEE (2013)
16. Xuan, K., Zhao, G., Taniar, D., Rahayu, W., Safar, M., Srinivasan, B.: Voronoi-based range and continuous range query processing in mobile databases. Journal of Computer and System Sciences **77**(4), 637–651 (2011)
17. Yu, X., Pu, K.Q., Koudas, N.: Monitoring k-nearest neighbor queries over moving objects. In: Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on, pp. 631–642. IEEE (2005)